Breviar

Laborator 3 - Clase și Obiecte în Java

- Responsabil: Mihai Nan [mailto:mihai.nan.cti@gmail.com]
- Profesor titular: Carmen Odubășteanu

Introducere

Presupunem că dorim *să descriem*, uzitând un limbaj de programare, un **obiect** carte. În general, o carte poate fi caracterizată prin titlu, autor și editură. Cum am putea realiza această descriere formală?

Dacă descriem acest obiect, tip abstract de date, într-un limbaj de programare structural, spre exemplu limbajul C, atunci vom crea, ca mai jos, o structură `Carte` împreună cu o serie de funcții cuplate de această structură. Cuplajul este realizat prin faptul că orice funcție care operează asupra unei cărți conține în lista sa de argumente o variabilă de tip `Carte`.

```
 typedef struct carte {
        char *titlu, *autor;
        int nr_pagini;
}*Carte;

void initializare(Carte this, char* titlu, char* autor, int nr_pagini) {
        this->titlu = strdup(titlu);
        this->autor = strdup(autor);
        this->nr_pagini = nr_pagini;
}

void afisare(Carte this) {
        printf("%s, %s - %d\n", this->autor, this->titlu, this->nr_pagini);
}
```

Dacă modelăm acest obiect într-un limbaj orientat pe obiecte (în acest caz, Java), atunci vom crea o **clasă** `Carte` ca mai jos.

```
class Carte {
    String nume, autor;
    int nr_pagini;

    public Carte(String nume, String autor, int nr_pagini) {
        this.nume = nume;
        this.autor = autor;
        this.nr_pagini = nr_pagini;
    }

    public Carte() {
        this("Enigma Otiliei", "George Calinescu", 423);
    }

    public String toString() {
        return this.autor + ", " + this.nume + " - " + this.nr_pagini;
    }

    public static void main(String args[]) {
        Carte carte;
        carte = new Carte("Poezii", "Mihai Eminescu", 256);
        System.out.println(carte);
    }
}
```

Se poate observa cu usurință, în cadrul exemplului de mai sus, că atât datele cât și **metodele** (funcțiile) care operează asupra acestora se găsesc în interiorul aceleiași entități, numită **clasa**. Evident, în codul din exemplu sunt folosite concepte care nu au fost încă explicate, dar cunoașterea și înțelegerea lor reprezintă scopul principal al acestui laborator.

Clase și Obiecte

Ce este un obiect? Ce este o clasă?

- Atunci când un producător crează un produs, mai întâi acesta specifică toate caracteristicile produsului într-un document de specificații, iar pe baza acelui document se crează fizic produsul. De exemplu, calculatorul este un produs creat pe baza unui astfel de document de specificații. La fel stau lucrurile și într-un program orientat pe obiecte: mai întâi se crează **clasa** obiectului (documentul de specificații) care înglobează toate caracteristicile unui **obiect** (instanță a clasei), dupa care, pe baza acesteia, se crează (instanțiază) obiectul în memorie.
- În general, putem spune că o clasă furnizează un șablon ce spcifică datele și operațiile ce aparțin obiectelor create pe baza șablonului - în documentul de specificații pentru un calculator se menționează că acesta are un monitor și o serie de periferice.

Programarea orientată pe obiecte este o metodă de implementare a programelor în care acestea sunt organizate sub formă unor colecții de obiecte care cooperează între ele, fiecare obiect reprezentând instanța unei clase.

Definirea unei clase

- Din cele de mai sus deducem că o clasă descrie un obiect, în general, un nou tip de date. Într-o **clasă** găsim **date** și **metode** ce operează asupra datelor respective.
- Pentru a defini o clasă, trebuie folosit cuvântul cheie `class` urmat de numele clasei.

```
class <class_name> {
    field;
    method;
}
```

O **metodă** nu poate fi definită în afara unei **clase**.

Datele `nume`, `autor`, `nr_pagini` definite în clasa `Carte` se numesc **atribute**, **date-membru**, **variabile-membru** sau **câmpuri**, iar operațiile `toString` și `main` se numesc **metode**.

Fiecare clasă are un set de **constructori** care se ocupă cu **instanțierea** (inițializarea) obiectelor nou create. De exemplu, clasa `Carte` are doi constructori: unul cu trei parametri și unul fără parametri care îl apelează pe cel cu trei parametri.

Crearea unui obiect

- Spuneam mai sus că un obiect reprezintă o instanță a unei clasa. În Java, instanțierea sau crearea unui obiect se face dinamic, folosind cuvântul cheie `new` și are ca efect crearea efectivă a obiectului cu alocarea spațiului de memorie corespunzător.
- Așa cum fiecare calculator construit pe baza documentului de specificații are propriile componente, fiecare obiect de tip `Carte` are propriile sale atribute.
- **Inițializarea** se realizează prin intermediul constructorilor clasei respective. Inițializarea este, de fapt, parte integrantă a procesului de instanțiere, în sensul că imediat după alocarea memoriei ca efect al operatorului `new` este apelat constructorul specificat. Parantezele rotunde după numele clasei indică faptul că acolo este, de fapt, un apel la unul din constructorii clasei și nu simpla specificare a numelui clasei.
- În Java, este posibilă și crearea unor **obiecte anonime**, care servesc doar pentru inițializarea altor obiecte, caz în care etapa de declarare a referinței obiectului nu mai este prezentă.

Declararea unui obiect nu implică alocarea de spațiu de memorie pentru acel obiect. Alocarea memoriei se face doar la apelul operatorului `new`.

```
class Point {
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Dimension {
    int width, height;

    public Dimension(int width, int height) {
        this.width = width;
        this.height = height;
    }
}

class Rectangle {
    Point p;
    Dimension d;

    public Rectangle(Point p, Dimension d) {
        this.p = p;
        this.d = d;
    }

    public static void main(String args[]) {
        Rectangle patrat = new Rectangle(new Point(0, 0), new Dimension(10, 10));
    }
}
```

Referințe la obiecte

- În secțiunea anterioară, am văzut cum se definește o clasă și cum se crează un obiect. În această secțiune vom vedea cum putem executa operațiile furnizate de obiecte. Pentru a putea avea acces la operațiile furnizate de către un obiect, trebuie să deținem o **referință** spre acel obiect.
- Odată un obiect creat, el poate fi folosit în următoarele sensuri: aflarea unor informații despre obiect, schimbarea stării sale sau executarea unor acțiuni. Aceste lucruri se realizează prin aflarea sau schimbarea valorilor variabilelor sale, respectiv prin apelarea metodelor sale.
- Declarea unei referințe numite `carte` spre un obiect de tip `Carte` se face în felul următor: `Carte carte;`

Faptul că avem la un moment dat o referință nu implică și existența unui obiect indicat de acea referință. Până în momentul în care referinței nu i se atașează un obiect, aceasta nu poate fi folosită.

Valoarea `null`, ce înseamnă **niciun obiect referit**, nu este atribuită automat tuturor variabilelor referința la declararea lor. Regula este următoarea: dacă referința este un membru al unei clase și ea nu este inițializată în niciun fel, la instanțierea unui obiect al clasei respective, referința va primi implicit valoarea `null`. Dacă însă referința este o variabilă locală ce aparține unei metode, inițializarea implicită nu mai funcționeaza. De aceea, se recomandă ca programatorul să realizeze **intotdeauna** o inițializare explicită a obiectelor.

- După cum am observat în exemplul oferit în prima secțiune, apelul metodei `toString` nu este `toString(carte)`, ci `carte.toString()` întrucât metoda `toString` aparține obiectului referit de `carte` - se apelează metoda `toString` pentru obiectul referit de variabila `carte`.
- Pentru o înțelegere mai bună a conceptului de referință a unui obiect, considerăm exemplul de mai jos în care creăm două obiecte de tip `Carte` precum și trei referințe spre acest tip de obiecte.
- Fiecare dintre obiectele `Carte` are alocată o zonă proprie de memorie, în care sunt stocate valorile câmpurilor `nume`, `autor`, `nr_pagini`. Ultima referință definită în exemplul de mai jos, `c3`, va referi și ea exact același obiect ca `c2`, adică al doilea obiect creat.

În cazul unui program, putem avea acces la serviciile puse la dispoziție de un obiect prin intermediul mai multor referințe.

```
class Test {
    public static void main(String args[]) {
        Carte c1 = new Carte("Poezii", "Mihai Eminescu", 326);
        Carte c2 = new Carte("Camil Petrescu", "George Calinescu", 426);
        Carte c3 = c2;
        c3.autor = "George Calinescu";
        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
    }
}
```

- Atribuirea `c3 = c2` nu a făcut altceva decât să atașeze referinței `c3` obiectul având aceeași identitate precum cel referit de `c2`, adică obiectul secund creat.

Trimiterea datelor în Java

Spre deosebire de C++, în Java nu există o modalitate prin care să poată fi făcută o diferențiere explicită între trimiterea parametrilor **prin referință** și trimiterea acestora **prin valoare**.

Conform specificației Java (secțiunea 4.3 [https://docs.oracle.com/javase/specs/jls/se9/html/jls-4.html#jls-4.3]), transmiterea tuturor datelor, atât a celor de tip obiect, cât și a celor primitive, este definită următoarea regulă:.

In Java argumentele sunt trimise doar **prin valoare** (pass-by-value).

Chiar dacă la o primă vedere această regulă poate să pară simplă, este necesară o explicație suplimentară. În cazul valorilor primitive, valoarea este considerată pur și simplu data asociată (exemple `1`, `10.5`, `true`) iar valoarea parametrilor este copiată de fiecare dată când ei sunt plasați în apeluri.

În ceea ce privește obiectele, în Java, se utilizează următoarea regulă, mai extinsă:

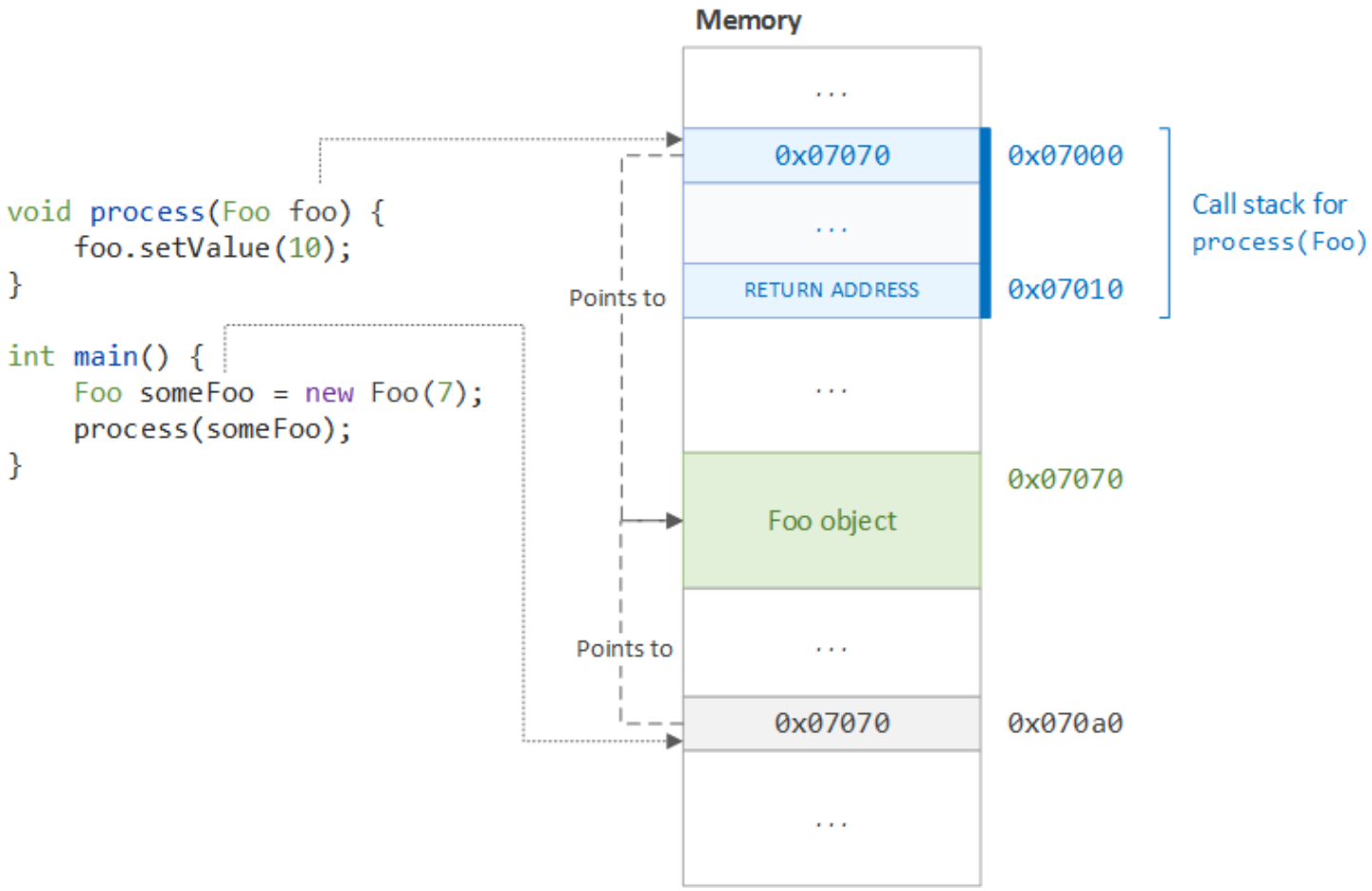
Valoarea asociată unui obiecte este, de fapt, un pointer, numit referință, la obiectul din memorie.

Spre exemplu, dacă definim o expresie de forma `Foo foo = new Foo();`, variabila `foo` nu deține obiectul `Foo` creat, ci, mai degrabă, o valoare a pointerului pentru obiectul `Foo` creat. Valoarea acestui pointer la obiect (ceea ce în specificația Java se numește **o referință de obiect** sau pur și simplu **referință**) este copiată de fiecare dată când obiectul este plasat ca argument al unui apel.

În Java, numai următoarele operații pot fi efectuate pe o referință de obiect:

- accesarea câmurilor;
- invocarea metodelor;
- operatorul pentru castare;
- operatorul pentru concatenarea string-urilor (atunci când primește ca parametru o referință la un obiect, o să realizeze o conversie a referinței la String, prin invocarea metodei `toString` pentru obiectul referențiat);
- operatorul `instanceof`;
- operatorii de egalitate pentru referințe: `==` și `!=`;
- operatorul condițional: `? :`.

În practică, acest lucru înseamnă că putem schimba câmpurile obiectului trimis ca parametru într-o metodă și să invocăm metodele acestuia, însă nu putem schimba obiectul spre care pointează referința. Deoarece referința este plasată prin valoare, pointerul original este copiat în stiva de apeluri atunci când metoda este invocată.



Pentru a înțelege mai bine conceptele prezentate în această secțiune, puteți consulta și analiza rezultatele pentru următoarele secvențe de cod.

```
int someValue = 10;
int anotherValue = someValue;
someValue = 17;
System.out.println("Some value = " + someValue);
System.out.println("Another value = " + anotherValue);
```

```
public class Test {
	public void process(int value) {
		System.out.println("Entered method (value = " + value + ")");
		value = 50;
		System.out.println("Changed value within method (value = " + value + ")");
		System.out.println("Leaving method (value = " + value + ")");
	}

	public static void main(String args[]) {
		Test processor = new Test();
		int someValue = 7;
		System.out.println("Before calling method (value = " + someValue + ")");
		processor.process(someValue);
		System.out.println("After calling method (value = " + someValue + ")");
	}
}
```

```
class Ball {}

class Main {
	public static void main(String args[]) {
		Ball someBall = new Ball();
		System.out.println("Some ball before creating another ball = " + someBall);
		Ball anotherBall = someBall;
		someBall = new Ball();
		System.out.println("Some ball = " + someBall);
		System.out.println("Another ball = " + anotherBall);
```

```
        }
}
```

```
class Vehicle {
    private String name;
    public Vehicle(String name) {
        this.name = name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    @Override
    public String toString() {
        return "Vehicle[name = " + name + "]";
    }
}

class VehicleProcessor {
    public void process(Vehicle vehicle) {
        System.out.println("Entered method (vehicle = " + vehicle + ")");
        vehicle.setName("A changed name");
        System.out.println("Changed vehicle within method (vehicle = " + vehicle + ")");
        System.out.println("Leaving method (vehicle = " + vehicle + ")");
    }
    public void processWithReferenceChange(Vehicle vehicle) {
        System.out.println("Entered method (vehicle = " + vehicle + ")");
        vehicle = new Vehicle("A new name");
        System.out.println("New vehicle within method (vehicle = " + vehicle + ")");
        System.out.println("Leaving method (vehicle = " + vehicle + ")");
    }
}

class Main {
        public static void main(String args[]) {
                VehicleProcessor processor = new VehicleProcessor();
                Vehicle vehicle = new Vehicle("Some name");
                System.out.println("Before calling method (vehicle = " + vehicle + ")");
                processor.process(vehicle);
                System.out.println("After calling method (vehicle = " + vehicle + ")");
                processor.processWithReferenceChange(vehicle);
                System.out.println("After calling reference-change method (vehicle = " + vehicle + ")");
        }
}
```

Componența unei clase

Clasele, așa cum am văzut deja, sunt definite folosind cuvântul cheie `class`. În următoarele secțiuni, vom vorbi despre diferite categorii de membri care pot apărea în interiorul unei clase.

Constructori

- În multe cazuri, atunci când instanțiem un obiect, ar fi folositor ca obiectul să aibă anumite atribute inițializate.
- Inițializarea atributelor unui obiect se poate face în mod automat, la crearea obiectului, prin intermediul unui **constructor**. Principalele caracteristici ale unui constructor sunt:
    1. un constructor are același nume ca clasa în care este declarat;
    2. un constructor nu are tip returnat;
    3. un constructor se apelează automat la crearea unui obiect;
    4. un constructor se execută la crearea obiectului și numai atunci.

Dacă programatorul nu prevede într-o clasa niciun constructor, atunci compilatorul va genera pentru clasa respectivă un constructor implicit fără niciun argument și al cărui corp de instrucțiuni este vid.

Dacă programatorul include într-o clasă cel puțin un constructor, compilatorul nu va mai genera constructorul implicit.

```
class Carte {
        private String autor, nume;

        public Carte(String autor, String nume) {
                this.autor = autor;
                this.nume = nume;
        }

        public String toString() {
                return "Autor: " + autor + "\n" + "Titlul: " + nume;
        }
}

class Creion {
        private String culoare;

        public String getCuloare() {
                return culoare;
        }
}

class Test {
        public static void main(String args[]) {
                Carte c1, c2;
                c1 = new Carte(); //EROARE, deoarece nu avem constructor de aritate 0
                c2 = new Carte("George Calinescu", "Enigma Otiliei"); //CORECT
                Creion c3;
                c3 = new Creion(); //CORECT, deoarece nu am definit niciun constructor => exista cel predefinit
        }
}
```

Membri statici

Atunci când definim o clasă, specificăm felul în care obiectele de tipul acelei clase arată și se comportă. Dar până la crearea efectivă a unui obiect, folosind new, nu se alocă nicio zonă de memorie pentru atributele definite în cadrul clasei, iar la crearea unui obiect se alocă acestuia memoria necesară pentru fiecare atribut existent în clasa instanțiată. Tot până la crearea efectivă a unui obiect nu putem beneficia de serviciile definite în cadrul unei clase. Ei bine, există și o excepție de la regula prezentată anterior - **membrii statici** (atribute și metode) ai unei clase. Acești membri ai unei clase pot fi folosiți direct prin intermediul numelui clasei, fără a deține instanțe pentru respectiva clasă.

Un membru static al unei clase caracterizează clasa în interiorul căreia este definit precum și toate obiectele clasei respective.

Un membru al unei clase (atribut sau metodă) este static dacă el este precedat de cuvântul cheie `static`.

Din interiorul unei metode statice pot fi accesați doar alți membri statici ai clasei în care este definită metoda, accesarea membrilor nestatici ai clasei producând o eroare de compilare.

Trebuie avut în vedere contextul static al metodei **main**. Dintr-un context static nu se pot apela funcții nestatice, în schimb, se pot crea obiecte ale oricărei clase.

Principii POO

Mai multe funcții pot avea același nume în același domeniu de definiție, dacă se pot diferenția prin numărul sau tipul argumentelor de apel.

Supraîncarcarea

În Java, se pot găsi două sau mai multe metode, în cadrul aceleiași clase, care să aibă același nume, atâta timp cât argumentele lor sunt diferite. În acest caz, se spune că metoda este supraîncărcată, iar procedeul se numește **supraîncarcarea metodelor**. Pentru o mai bună înțelegere a acestui principiu POO, se va oferi, în continuare, un exemplu pentru o metodă care determină maximul.

```
class Test {
    public int maxim(int a, int b) {
        if(a > b)
            return a;
        else
            return b;
    }

    public int maxim(String s1, String s2) {
        if(s1.compareTo(s2) < 0)
            return 2;
        else
            return 1;
    }

    public int maxim(int a, int b, int c) {
        if(maxim(a, b) < c)
            return c;
        else
            return maxim(a, b);
    }
}
```

Un alt exemplu elocvent, pentru acest prinicpiu POO, este operatorul + care execută operații diferite în cotexte diferite.